StateFormer: Fine-Grained Type Recovery from Binaries using Generative State Modeling

Kexin Pei
kpei@cs.columbia.edu
Columbia University
New York, USA

Jonas Guan
jonas@cs.toronto.edu
University of Toronto
Toronto, Canada

Matthew Broughton
mb4207@columbia.edu
Columbia University
New York, USA

Zhongtian Chen
zc2399@columbia.edu
Columbia University
New York, USA

Songchen Yao
sy2743@columbia.edu
Columbia University
New York, USA

David Williams-King
dwk@cs.columbia.edu
Columbia University
New York, USA

Vikas Ummadisetty
ummadisettyvikas@gmail.com
Dublin High School
Dublin, USA

Junfeng Yang
junfeng@cs.columbia.edu
Columbia University
New York, USA

Baishakhi Ray
rayb@cs.columbia.edu
Columbia University
New York, USA

Suman Jana
suman@cs.columbia.edu
Columbia University
New York, USA

ABSTRACT

Binary type inference is a critical reverse engineering task supporting many security applications, including vulnerability analysis, binary hardening, forensics, and decompilation. It is a difficult task because source-level type information is often stripped during compilation, leaving only binaries with untyped memory and register accesses. Existing approaches rely on hand-coded type inference rules defined by domain experts, which are brittle and require nontrivial effort to maintain and update. Even though machine learning approaches have shown promise at automatically learning the inference rules, their accuracy is still low, especially for optimized binaries.

We present StateFormer, a new neural architecture that is adept at accurate and robust type inference. StateFormer follows a two-step transfer learning paradigm. In the pretraining step, the model is trained with Generative State Modeling (GSM), a novel task that we design to teach the model to statically approximate execution effects of assembly instructions in both forward and backward directions. In the finetuning step, the pretrained model learns to use its knowledge of operational semantics to infer types.

We evaluate StateFormer's performance on a corpus of 33 popular open-source software projects containing over 1.67 billion variables of different types. The programs are compiled with GCC and LLVM over 4 optimization levels `O0`-`O3`, and 3 obfuscation passes based on LLVM. Our model significantly outperforms state-of-the-art ML-based tools by 14.6% in recovering types for both function arguments and variables. Our ablation studies show that GSM improves type inference accuracy by 33%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468607

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; • **Computing methodologies → Machine learning**.

KEYWORDS

Type Inference, Reverse Engineering, Transfer Learning, Machine Learning for Program Analysis

ACM Reference Format:
Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: Fine-Grained Type Recovery from Binaries using Generative State Modeling. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468607

1 INTRODUCTION

Recovering source-level data types from binaries is very useful for many security-critical software engineering tasks, such as vulnerability analysis [18, 37, 45], binary hardening [30, 53, 57, 74, 96, 99, 101], memory introspection [43, 89], and decompilation [4, 26]. Type inference in binaries involves reconstructing source-level constructs, such as local function variables and data types, from untyped byte-addressed memory and registers. This process is challenging because the reconstruction is based on incomplete information – most source-level information is stripped during compilation for optimization and to deter reverse engineering.

Traditional approaches to type inference rely extensively on hand-coded rules defined by domain experts. These rules facilitate (1) *recognizing* types directly from specified patterns (*e.g.,* consecutive printable characters for detecting strings); and (2) *propagating* types from known type sinks (*e.g.,* known string manipulation functions) to registers and memory regions storing the source-level variables. Unfortunately, these rules are brittle [16, 66] and require continuous effort to adapt to new instruction sequences introduced by compiler and architecture evolution [10].

As a result, recent years have witnessed a growing interest in data-driven approaches leveraging Machine Learning (ML) for binary type inference [20, 40, 59]. These approaches mitigate the reliance on hand-coded heuristics by learning from a rich training set of diverse binaries. Moreover, their learned representations have been shown to generalize across various compilers, operating systems, and architectures and are highly efficient to compute (*i.e.,* the underlying learning algorithms are amenable to GPU parallelization).

While promising, existing ML-based approaches still cannot recover data types with high accuracy or robustness, especially in the presence of compiler optimizations [20, 40]. The types are essentially abstractions describing *how a data object is expected to be manipulated and used during execution*. Therefore, the inherent challenge faced by all ML-based approaches is to understand the effects of the runtime execution of instructions in the target binary [55, 83], *i.e.,* the *operational semantics* of code blocks [64]. For example, on x64, the runtime effect of iterative increments of the `rcx` register by 1 together with the instruction `mov rax,[rdx+rcx*4]` is indicative of traversing an `int` array.

Unfortunately, existing ML-based approaches are agnostic to the execution effects of code as they learn the direct mapping between *static* code tokens and corresponding types in an end-to-end fashion. A model trained this way often learns spurious correlations [7], taking shortcuts to leverage simple yet brittle patterns for inferring types. For example, Chua *et al.* [20] showed that their model, EKLAVYA, mispredicts the type of the argument to the function `ck_fopen` from Diffutils to be integer instead of pointer. A completely unrelated instruction within `ck_fopen` (namely `callq 0x3fc`) contributes the most to the misprediction. Without understanding how an integer is accessed and manipulated during execution and the effects of `callq`, EKLAVYA establishes a spurious correlation that the internal call instruction implies an `int` argument to `ck_fopen`.

In this paper, we present StateFormer, a new neural architecture that explicitly learns the operational semantics of assembly for type inference. Specifically, we design a novel pretraining task to teach the StateFormer model the operational semantics of both *data and control* flow behavior of diverse code blocks, and then finetune the pretrained model for type inference.

Learning operational semantics. A human reverse engineer often makes sense of a target binary by following its assembly instructions through *mental simulation* of their execution. While the reverse engineer might not accurately resolve all invoked branches by following control flow or compute the precise values of all states by following data flow during simulation, she can still get a rough idea of what the code does by approximately following the operational semantics of code blocks. Our key insight is to teach the StateFormer model, via a novel pretraining task, the approximate operational semantics of assembly by forcing the model to predict how different sequences of instructions *transform the underlying program states*. Specifically, the pretraining task asks the model to predict the changed values of registers and memory after executing each instruction, which captures the operational semantics of assembly code [29, 64, 70]. This gives the model an understanding of the execution effects of code, which helps the model to infer the types of low-level registers and memory regions based on the instructions used to manipulate them without executing any parts of the code during inference.

Generative State Modeling. We design a novel pretraining task, Generative State Modeling (GSM), where we train a neural network to reconstruct the *complete set* of its execution states while taking the assembly code and a *very small subset* of its execution states (*e.g.,* register values at specific program points) as input. For example, given the instruction sequence: `inc ecx;add ecx,3;xor ecx,ecx;mov ebx,ecx;` and its corresponding execution states `ecx=0;ecx=1;ecx=4;ecx=0;(ebx=0,ecx=0)`, we feed the model with all the instructions and only the execution state after the second instruction, *i.e.,* `ecx=4`. Our training process forces the model to compute all the preceding and succeeding states: `ecx=0;ecx=1;` and `ecx=0;(ebx=0,ecx=0)`. Therefore, to achieve low loss on the GSM task, the model needs to understand the operational semantics of `inc`, `add`, `xor`, and `mov`.

GSM dynamically selects random subsets of states as inputs across different training samples and iterations. Moreover, GSM is fully self-supervised [24], implying that we can collect data from an unrestricted number of binaries found in the wild. As a result, GSM creates diverse prediction tasks that compel the model to approximately reason about the effects of both *data* and *control* instructions, in both *forward* and *backward* directions – a critical capability for type recognition and propagation [55, 83]. During pretraining with GSM, StateFormer encodes such a reasoning capability as part of its network parameters, known as embeddings. Such embeddings can then be *finetuned* for type inference as a finetuning task with a few binaries with labeled types.

Consider again the example of inferring the traversal of an `int` based on iterative increments of the `rcx` register by 1 together with the instruction `mov rax,[rdx+rcx*4]`. The output of pretrained StateFormer will be a sequence of embeddings encoding the effects of `inc`, `mov` on other registers and memory locations. Therefore, instead of training on raw code sequences from scratch, the finetuning process can easily exploit the learned execution effects of code compressed in these embeddings to predict that `rdx` contains the base address of an `int` array.

StateFormer neural architecture. To efficiently pretrain with GSM, we develop a novel neural architecture specifically designed for learning operational semantics of assembly instructions. First, as the model takes as input both the program code and program states, we develop a *multi-modal encoding module* that can be trained on heterogeneous inputs in different formats.

Second, we construct two explicit objective functions to jointly optimize the model to understand the operational semantics of both *data flow* and *control flow*. Specifically, to help the model learn about control-flow, which requires learning operational semantics of comparison instructions (*e.g.,* `cmp`), we annotate the non-executed paths with dummy program states to incorporate predicting non-executed paths as a part of the StateFormer's pretraining task. To help the model to better understand the operational semantics of data flow, which often involves assignment (*e.g.,* `mov`) and arithmetic instructions (*e.g.,* `add`) on numerical values, we explicitly model the numerical values in both decimal and hexadecimal formats with a trainable *numerical representation module* based on the neural arithmetic unit (NAU) [58].

Finally, as the composite execution effects of a piece of code can result from the interactions between faraway instructions, we leverage self-attention layers from Transformer [93], which is amenable to learning long-range dependencies without manually constructing the dependencies (*e.g.,* graph neural net [63, 97]). We show in Section 5.5 that such a design indeed achieves high testing accuracy in GSM for unseen program state traces.

Result summary. We evaluate STATEFORMER on a corpus of 33 popular open-source software projects with 1.67 billion source variables of different types. The programs are compiled for 4 instruction set architectures (x86, x64, ARM, and MIPS), by 2 compilers (GCC and LLVM), and with 4 optimization levels (`O0`-`O3`) and 3 obfuscation passes based on LLVM [104]. By training with GSM, our model outperforms the state-of-the-art ML-based tools by up to 14.6% in recovering types for both function arguments and variables. Our extensive ablation studies show that STATEFORMER trained with GSM substantially boosts the type inference accuracy by 33%. We make the following contributions.

- We propose a new pretraining task, Generative State Modeling (GSM), to explicitly learn the operational semantics of assembly code for accurate and robust type inference.
- We develop a novel neural architecture, STATEFORMER, with specially designed sub-modules to learn the operational semantics of both data flow and control flow instructions.
- We evaluate STATEFORMER on an extensive collection of 33 open-source software projects across different architectures, compilers, optimizations, and obfuscations. After training with GSM, STATEFORMER outperforms the state-of-the-art learning-based tools by 14.6%. Our ablation studies unveil that training with STATEFORMER boosts the type inference accuracy by 33%. We release the code and datasets of STATEFORMER at https://github.com/CUMLSec/stateformer.

2 OVERVIEW

The high-level workflow of STATEFORMER follows the general transfer learning paradigm. As shown in Figure 1, we first pretrain STATEFORMER with GSM by training it to reconstruct the masked states (grayed-out) in the trace of program states of various assembly instructions (Section 2.3). We train STATEFORMER to reconstruct both the data and control states (Section 2.4). After pretraining STATEFORMER with GSM, we transfer its learned knowledge by finetuning on the type inference task (defined in Section 3.4).

2.1 Problem Definition

We consider the problem of mapping untyped low-level registers or memory regions (specified by memory offsets) to the corresponding source-level types. The source-level types are associated with function arguments, local, static, and global variables. The granularity of recovered source-level types varies widely across existing works [16], ranging from primitive (*e.g.,* `int`, `float`) and aggregate (*e.g.,* `struct`, `array`) types to classes in object-oriented programs and recursive types such as trees and lists.

We focus on the standard C primitive, aggregate, and pointer types. Our supported types are more *fine-grained* than prior works [20, 40, 59], which only support a strict subset of ours (see Section 3.4 for the complete list of types we support). Predicting fine-grained types, while very helpful to the human reverse engineer to better understand the target binary, is a challenging learning task that must distinguish between subtly different access patterns of different types [83].

We formulate type inference as a classification task. Specifically, given a sequence of assembly instructions, STATEFORMER predicts the type labels for each operand in the instructions. Note that STATEFORMER performs the type prediction in *one shot* (see Section 3 for design specifics), as opposed to the traditional type propagation approaches that infer the types one-by-one in a sequence of instructions. As we show in Section 5.3, this design brings significant performance gains during inference.

2.2 Understanding Operational Semantics Helps Type Inference

While reverse engineering types from binaries, human analysts need to understand what a target function computes without actually executing the binary. Often the analyst follows the assembly instructions by simulating the execution in their mind. Without knowing the exact initial program state during the function call, the analyst cannot accurately resolve the taken branches or compute the precise value of all states during the simulation. Still, they can get a rough idea of what the code does. This loose approximation of the operational semantics of the code allows the understanding of its runtime behavior, providing strong hints about the underlying data types [83].

For instance, given a pointer `a`, observing a dereference like `*(a+4)` in the execution behavior might imply a 4-byte read of the object at `a`, indicating an `int` or a pointer type on 32-bit systems. Similarly, contiguous dereferencing of sequential addresses like `*(a)`, `*(a+1)`, ... suggests that `a` is likely an array of `char`s. Examining precise runtime behaviors of a binary over many inputs with high-coverage dynamic analysis is prohibitively expensive [51, 83, 87], Therefore, in this paper, we use ML models to learn approximate operational semantics of binaries in a data-driven manner and use this knowledge to *statically* infer types.

2.3 Learning Operational Semantics with GSM

Our key motivation for developing GSM is to teach an ML model to approximate operational semantics of code, *i.e.,* its execution effects, which we then exploit for type inference. Teaching an ML model the code execution effects is challenging due to many possible combinations of instructions that introduce complex data and control flow dependencies. Therefore, it is not practical to manually engineer input features or target labels to represent the execution effects and train the model to understand them. To this end, GSM explores a self-supervised approach that exploits a large number of traces that can be cheaply generated from many code blocks using under-constrained dynamic execution, *e.g.,* micro-execution (detailed in Section 3.1), to *automate* learning diverse instructions' execution effect with a carefully-designed training task.

Predicting masked states. The training task performed in GSM requires a neural network to reconstruct the whole micro-execution traces (*i.e.,* all recorded program states) in the training data based on the corresponding code blocks. To learn on a huge number of traces, we exploit stochasticity to efficiently train a network for GSM. Specifically, for each training sample in each epoch, we randomly mask (*i.e.,* remove) some states in the traces. Such randomness

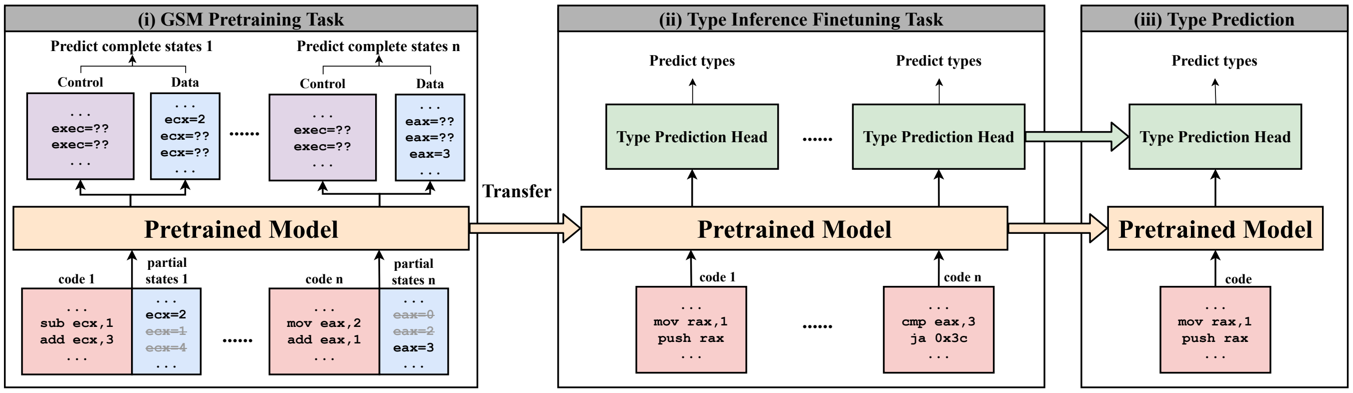


Figure 1: StateFormer workflow. We first pretrain StateFormer with GSM. We then stack new type prediction heads on top of the pretrained model and finetune both the pretrained model and the stacked heads for type inference. Finally, the finetuned model will only take the program code as input (we do not execute the code during type inference) and predict the type.

ensures that the model cannot consistently achieve low loss by taking shortcuts that only work well for a few states, traces, or code blocks.

While deciding which states to mask, GSM does not follow the sequential execution order of states as recorded in the traces. This design choice ensures that the model learns to reason about both forward and backward execution effects of a diverse set of code blocks. Understanding these forward and backward dependencies is known to be crucial for accurate type inference [55, 83].

Difference with masked language models. While our stochastic masking setup is inspired by the Masked Language Modeling (MLM) used in learning natural language semantics [24], the key difference is that natural languages are not stateful, *i.e.,* they have no notion similar to program execution. Therefore, the model trained by MLM only uses unmasked words in the *neighboring context* to predict the masked words, exploiting the common *local word phrase patterns*. While in GSM, the unmasked states alone provide little observable patterns due to high masking rate – the model *has to also look at the corresponding instructions*, understand their execution effects on the unmasked states, in order to correctly predict the masked states.

2.4 StateFormer Architecture

Learning instruction-state dependencies. Achieving low loss on GSM, by design, requires a neural network to understand the long-range dependencies between instructions and unmasked program states. However, standard fully-connected or recurrent networks are inefficient at learning long-range dependencies between different parts of the network inputs [61, 93, 98].

To avoid these issues, we develop a hierarchical input embedding module to learn the interactions between program states and instructions. Specifically, we design two input sub-networks for learning two embeddings of the binary code and traces – one for the registers and instruction opcodes and another for the concrete data values. We combine these representations by aggregating the embeddings with a vector addition operation and feeding them into a sequence of self-attention layers that facilitate capturing long-range dependencies [93] (Section 3.2). Finally, we use two output sub-networks to decode the output of self-attention layers for two different objectives: (1) regression for predicting the program data state and (2) classification for predicting the program control state (Section 3.3).

Learning representations for numerical values. Typical embeddings for numerical tokens (*i.e.,* register values) – just like how any discrete token is embedded – are known to fail to extrapolate to unseen values even on the outputs of simple arithmetic operations like addition [88]. As understanding data and control flow often requires reasoning the execution effect of arithmetic instructions (*e.g.,* `add rax, rbx`), we use Neural Arithmetic Units (NAUs) [58] as part of the subnetwork for data value embeddings. Note that our NAU layers, unlike the original NAU model that directly takes numerical values as input, learn to represent the numerical values (both decimal and hexadecimal formats) as embeddings. We have done a thorough study and refer interested readers to our supplementary material.

3 METHODOLOGY

We now provide the details of our methodology, including how we collect runtime states of binary programs, the architecture of StateFormer, and how we distill the learned knowledge in StateFormer from training GSM for type inference.

3.1 Collecting Program States

To train StateFormer with GSM, we need to obtain runtime execution traces of binary programs. Ideally, we want to collect diverse traces with different instructions and control flow to learn miscellaneous operational semantics for type inference in various scenarios. However, the typical dynamic analysis approach is often limited by path coverage, resulting in potentially restricted sets of covered instructions. Therefore, we adopt micro-execution [33] to support tracing arbitrary parts of a binary program without having to find concrete program inputs that maximize coverage.

Without executing the program from its entry point, our execution engine needs to initialize intermediate program states (*i.e.,* registers and memory content) with randomized values, which can be under-constrained (*i.e.,* infeasible when executing the program normally). In addition, we focus on program states that are *explicitly manipulated* in instructions (*e.g.,* we only log the value of `eax` in

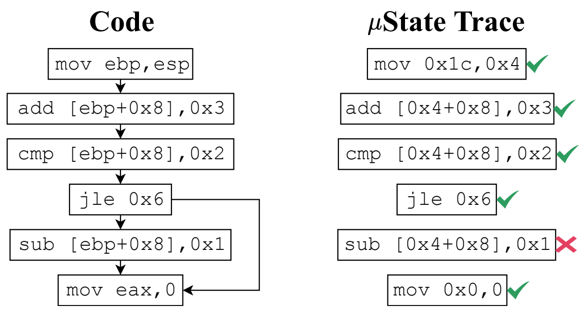


Figure 2: Sample μState trace consisting of both the data states associated with each instruction and the control states (indicated by ✓and ✗) generated by the micro-execution.

`sub eax,1`, instead of logging all registers, flags, and values in memory). Therefore, we call our collected program states as *partial states* (μState), implying that they might differ from the genuine program states from actual program executions.

μState collection. We collect μState traces and mask a random subset of them to train the model to reconstruct the complete μState trace. μState consists of two sources of information. (1) The concrete values of all registers, memory addresses, and hardcoded offsets that appear in the instruction, dubbed μDataState. (2) The boolean annotation indicating whether each instruction in the code is executed in a given μState, dubbed μControlState. The former appears in both StateFormer's input and output (*i.e.,* subset of μDataState as input, complete set of μDataState as output). The latter appears only in StateFormer's output. Section 3.2 elaborates on how these two parts of μState are used to train StateFormer.

Figure 2 shows an example of μState trace generated by micro-executing a simple code block, *e.g.,* the concrete values of registers are μDataState and the ✓and ✗ besides each instruction indicates μControlState. We assign dummy values (`$$`) to all opcode, as they do not hold any value during micro-execution. This helps to align μState and the assembly code sequence, which makes it convenient for StateFormer to aggregate them as network inputs (Section 3.2). To construct μControlState, we annotate each instruction with a binary indicator to denote whether it is executed or not.

3.2 StateFormer Input and Output

We construct 5 sequences for StateFormer input, namely (1) static assembly code sequence, (2) μDataState sequence, (3) instruction position sequence, (4) opcode/operand position sequence, and (5) architecture sequence. Each token of the 5 sequences are aligned and embedded into an embedding (a learned low-dimensional vector) with the same dimensions, such that they can be easily aggregated (*i.e.,* summation) as a single sequence of n embeddings: $x = \{x_1, ..., x_n\}$. Figure 3 illustrates an example input of StateFormer when training on GSM.

Encoding assembly code. The assembly code sequence with length n: $c = \{\texttt{add}, \texttt{ebp}, ...\}^n$ is constructed by tokenizing the assembly instructions from disassembled binaries. Besides treating both opcodes and operands as tokens, we keep punctuations as they provide crucial contextual hints, *e.g.,* the comma delimits the source and destination, and brackets indicate a dereference of a memory address.

Assembly code can have concrete numerical values hardcoded in instructions, which leads to a prohibitively large vocabulary size (*e.g.,* 2^{32} possible values in x86), making it challenging to embed all tokens in c. Therefore, we place the concrete value into the μDataState sequence and replace all numerical values (in both hexadecimal and decimal forms) with a special token `hex`. This reduces the vocabulary size of c across all instruction set architectures to only 648. We describe how we encode the numerical values in the following.

Encoding μDataState. We normalize μDataState sequence v as a two-dimensional array $v = \mathbb{V}^{n\times 8}$, where $\mathbb{V} = \{\texttt{0x00}, ..., \texttt{0xff}\} \cup \{\texttt{\$\$}\}$ (the union of 256 bytes and a dummy token `$$`). Each μDataState v_i can thus be viewed as a sequence of 8-byte tokens $\mathbb{V}^8$, where we transform all the numerical values into an 8-byte hexadecimal representation. For example, Figure 3 shows that a μDataState `0x6` is padded to (`00,00,00,00,00,00,00,06`). As each v_i is aligned with each token c_i in code sequence, we put 8 `$$`s for those c_i that do not have dynamic values (*e.g.,* opcode).

Such a setting reduces the vocabulary size used to encode all possible numerical values from 2^{64} (assume 64-bit architectures) to only 257. Moreover, representing a numerical value with fixed dimensions makes it easy to stack a single learnable module (see Section 3.3) to compute inter-dependencies between digits, learning useful hierarchical knowledge (*i.e.,* an address `0x104c` might be decomposed as a section base at `0x1000` with the offset `0x4c`).

Encoding spatial information and syntactic hint. As we flatten and concatenate all the assembly instructions as a plain code token sequence, the instruction boundaries and the relative location of tokens within each instruction become ambiguous. To this end, we introduce two positional encodings [93], namely the instruction positional encoding and opcode/operand positional encoding. The resulting instruction position sequence $p = \mathbb{Z}_+^n$ and opcode/operand position sequence $o = \mathbb{Z}_+^n$ annotate each token in c with their instruction position and the opcode/operand position within each position, respectively. Figure 3 shows the example of p and o.

When training with GSM, we mix the training samples from different instruction set architectures, which introduce disparate syntax in their assembly code. We thus append the architecture sequence a to indicate the architecture, which assists the model to transfer the learned instruction semantics useful on one architecture to another (*e.g.,* `push eax` in x86 has the similar semantics to `addi $sp,$sp,-4;sw $t0,($sp)` in MIPS) [49].

StateFormer output. StateFormer have different outputs depending on the training tasks. When it is in the pretraining stage with GSM, its output consists of complete μState trace including both μDataState trace and μControlState trace. We describe how these outputs participate in the computation of loss functions in Section 3.3. When we finetune StateFormer for type inference, its output is the prediction of type labels defined in Section 3.4.

3.3 Pretraining with GSM

Numerical representation module. We treat each value v_i in μDataState trace v as an 8-byte sequence. To learn the inter-dependencies between high and low bytes in v_i, we develop a

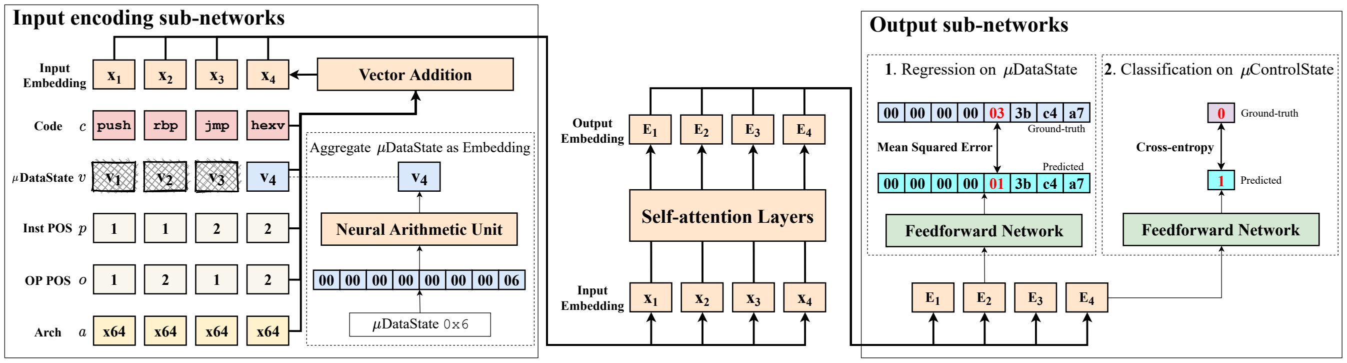


Figure 3: StateFormer's input-output when training with GSM and its architecture (we keep the color consistent with that of Figure 1). StateFormer takes as input the code sequence and a subset of μDataState (*e.g.,* v_4 in the figure). The other input sequences are described in Section 3.2. The loss functions measure (1) Mean Squared Error (MSE) between the reconstructed μDataState and the ground truth, and (2) Binary Cross-Entropy (BCE) between the predicted μControlState and the ground truth.

learnable neural module with Neural Arithmetic Unit (NAU) [58], which is shown beneficial to capture the semantics of numerical values involved in arithmetic operations (Section 2.4). Formally, let $v_i = (v_{i1}, ...v_{i8})$ denote the 8-byte sequence of v_i, we denote the aggregated embedding E_{v_i} as the representation of each μDataState: $E_{v_i} = NAU(Emb(v_{i1}), ..., Emb(v_{i8}))$, *e.g.,* $Emb(v_{i1})$ denote applying the embedding to the first byte token of v_i. Figure 3 briefly illustrates how a μDataState `0x6` gets encoded by NAU. Note that v_i in Figure 3 indicates the embedding E_{v_i}.

Sampling subset of μDataState. We sample a random subset of μDataState and replace them with `<MASK>` (*e.g.,* the grayed-out tokens as shown in Figure 3) tokens in the model input so that the model is trained to reconstruct the removed μDataState. We define P_{mask} as the percentage of the masked μDataState and study the effect of different P_{mask} on type inference in Section 5.4.

Multimodal encoding module. We only apply NAU to each μDataState sequence v. For other sequences, we apply regular embeddings. We end up with 5 embeddings for each token in each sequence: E_{c_i}, E_{v_i}, E_{p_i}, E_{o_i}, E_{a_i}. We then compute the vector sum of 5 embeddings and output a single embedding x_i: $x_i = sum(E_{c_i}, E_{v_i}, E_{p_i}, E_{o_i}, E_{a_i})$. The vector sum operation aggregates the multiple modalities (*e.g.,* instruction and state) of each token into a single embedding. When we compute attentions between these embeddings, *i.e.,* dot product [93], the cross-modality (instruction-state) dependencies are naturally computed, following the distributive property of multiplication: $x_i \cdot x_j = E_{c_i} \cdot E_{c_j} + E_{c_i} \cdot E_{v_j} + ... + E_{a_i} \cdot E_{a_j}$.

Loss functions. After encoding all the input sequences as a single sequence of embeddings $x = (x_1, x_2, .., x_n)$, we feed x to self-attention layers. The output of self-attention layers are known as the contextual embeddings $e = (e_1, e_2, .., e_n)$. We stack two independent 2-layer feedforward networks h_v and h_f, that takes e as input and output the predicted μDataState and μControlState. Formally, let $f = \{0, 1\}^n$ denote the μControlState labels, and M a set of locations in the masked μDataState v. We define the pretraining objective as:

$$min \sum_{i \in M} MSE(v_i, h_v(e_i)) + \alpha \sum_{i=1}^{n} BCE(f_i, h_i(e_i)) \qquad (1)$$

The first part of the objective function aims to minimize the Mean Squared Error (MSE) between the predicted 8-byte and the groundtruth 8-byte for the only masked μDataState. Note that MSE treats the output byte tokens as numerical values (as opposed to categorical as treated in the input). Such a setting encourages the loss to penalize predictions far from the groundtruth (*e.g.,* predicts `0x00` but the groundtruth is `0xff`). The second part of the objective function aims to minimize the Binary Cross-Entropy (BCE) between the predicted μControlState and the groundtruth, for all input tokens. α is the weighting hyperparameter that keeps the scale of both losses at roughly the same magnitude. As all the modules of StateFormer are differentiable, *i.e.,* NAU, FFN used for aggregating input sequences, self-attention layers, and h_v and h_i, optimizing Equation 1 can be efficiently solved by gradient descent.

3.4 Transfer Learning Type Inference

After pretraining with GSM, we transfer StateFormer's learned knowledge by finetuning it for type inference. We define our considered types in Figure 4, which serves as the labels for StateFormer to predict. Notably, our considered types are much more fine-grained than the existing ML-based type inference approaches. For example, EKLAVYA [20] does not distinguish signedness of the primitive types. Debin [40] does not handle floating point. And both works treat the pointer as a single type (`ptr`), without inferring what the pointer refers (*e.g.,* predicting `char*` or `struct*`).

As discussed in Section 2.3, we do not collect μState by micro-executing the code during finetuning. Specifically, we replace each token in v with the dummy token `$$` (described in Section 3.2) and still follow the same steps to compute the embeddings x. We then stack a new prediction head h_{type}, a 2-layer feedforward network, that takes as input e (the output of the last self-attention layers), and predicts the type labels defined in Figure 4 for each input code token. Formally, let t_i denote the groundtruth type of code token c_i, the objective function of finetuning task is defined as the Cross-Entropy between the predicted type $h_{type}(e_i)$ and t_i for each token in an input sequence with length n: $min \sum_{i=1}^{n} CE(t_i, h_{type}(e_i))$. During

⟨*type*⟩	::=	⟨*access*⟩ | '`no-access`'
⟨*access*⟩	::=	⟨*prim*⟩ | ⟨*agg*⟩ | ⟨*ptr*⟩
⟨*ptr*⟩	::=	⟨*prim*⟩ '`*`' | ⟨*agg*⟩ '`*`' | '`void*`'
⟨*prim*⟩	::=	'`float`' | '`double`' | '`long double`' | ⟨*sign*⟩ '`char`' | ⟨*sign*⟩ '`short`' | ⟨*sign*⟩ '`int`' | ⟨*sign*⟩ '`long`' | ⟨*sign*⟩ '`long long`'
⟨*agg*⟩	::=	'`struct`' | '`union`' | '`enum`' | '`array`'
⟨*sign*⟩	::=	'`signed`' | '`unsigned`'

Figure 4: The types that StateFormer predicts as output. We define the type hierarchy using the production rules for clarity, but we concretize all types during prediction, resulting in 35-type labels. $\langle prim \rangle$, $\langle agg \rangle$, and $\langle ptr \rangle$ stand for primitive, aggregate, and pointer types.

finetuning, both h_{type} and the pretrained model weights will be updated by gradient descent.

4 IMPLEMENTATION AND SETUP

We implement StateFormer using the `Fairseq` toolkit [65] based on PyTorch 1.6.0. All the experiments are run on a Linux server with Ubuntu 18.04, Intel Xeon 4214 at 2.20GHz with 48 virtual cores, 188GB RAM, and 4 Nvidia RTX 2080-Ti GPUs. To obtain ground-truth types for training and testing, we compile all the software projects with debugging information and parse the DWARF sections using `pyelftools` [12] and `Ghidra` [1].

μState collection. To log the program states (μState) for pretraining StateFormer on GSM task, we implement micro-execution using Unicorn [76], a cross-architecture CPU emulator based on QEMU [11]. Specifically, we micro-execute each function binaries (collected from the datasets described below) 9 times with different randomized initial values for registers and memory, generating 9 sets of μState for each function binary. To align the μState with the corresponding assembly instructions (Section 3.2), we leverage `Capstone` [75] to disassemble the function binaries.

Metrics. As described in Section 2, we treat type inference as a classification task. As the datasets have highly *imbalanced labels*, where the majority of tokens do not possess any type, we use precision (P), recall (R), and $F1$ score to measure the actual performance of StateFormer and all other tools. Let TP denote the number of correctly predicted types, FP denote that of incorrectly predicted types, TN denote the number of correctly predicted `no-access`, and FN denote the number of incorrectly predicted `no-access`. $P = TP/(TP + FP)$, $R = TP/(TP + FN)$, and $F1 = 2 \cdot P \cdot R/(P + R)$.

Baseline tools. We compare StateFormer with 3 state-of-the-art ML-based type inference prototypes: EKLAVYA [20], Debin [40], and TypeMiner [59]. These tools have been demonstrated to outperform traditional type inference techniques. For example, EKLAVYA has been shown to outperform TypeArmor [91], which is based on principled dataflow analysis such as def-use and liveness analysis.

EKLAVYA implements the function signature recovery task. The authors define the task as predicting the type of function arguments. Since EKLAVYA does not release their trained model, we use their reported numbers and use the same datasets to evaluate StateFormer's accuracy in recovering types for function argument.

Table 1: The statistics of our datasets, categorized by architecture (Arch), optimization (OPT), and obfuscation (OBF).

ARCH	OPT/OBF	# Variables	# Instructions	# Functions
ARM	O0	26,173,242	12,511,100	821,191
	O1	27,845,108	9,346,292	874,595
	O2	27,829,459	9,279,857	898,930
	O3	28,143,646	10,114,915	942,138
	Total	109,991,455	41,252,164	3,536,854
MIPS	O0	13,474,083	14,096,871	602,699
	O1	15,081,503	10,559,297	652,769
	O2	15,146,769	10,170,866	678,577
	O3	15,457,561	11,021,417	721,519
	Total	59,159,916	45,848,451	2,655,564
x86	O0	187,621,379	53,057,850	6,735,347
	O1	189,217,168	51,024,118	6,787,678
	O2	189,220,382	51,410,490	6,810,321
	O3	189,554,035	52,275,998	6,853,561
	Total	755,612,964	207,768,456	27,186,907
x64	O0	184,390,034	40,286,578	6,599,662
	O1	186,140,724	38,196,269	6,656,821
	O2	186,114,113	38,355,719	6,679,632
	O3	186,425,557	39,179,302	6,723,296
	bcf	714,892	12,960,798	119,706
	cff	644,018	11,604,224	90,740
	sub	714,310	6,960,835	119,481
	Total	745,143,648	187,543,725	26,989,338
Total		1,669,907,983	482,412,796	60,368,663

Debin recovers both variable types and names. As we do not study recovering source-level variable names but focus on obtaining variable types, we compare with Debin's type prediction only. Since Debin has released their trained model, we run Debin on our datasets directly and compare against its attained accuracy.

TypeMiner considers much finer-grained type labels than the previous two works. For example, it further distinguishes the pointer type to `struct` and `char`, while the former two do not. As TypeMiner is not open-sourced, we have contacted the authors to obtain their reported F1 scores and compare them to StateFormer by running StateFormer on their dataset.

These tools vary in their definition of the target types (*e.g.,* EKLAVYA is limited to predicting only function argument types) and the evaluated architectures (*e.g.,* TypeMiner only handles x64, EKLAVYA handles x86 and x64). Hence, we adjust our setup accordingly when comparing with the baselines.

Dataset. We collect 33 open-sourced software projects in their latest versions, including popular and large projects such as OpenSSL, ImageMagic, and Coreutils. Due to the page constraints, we put the details of the datasets in our supplementary material. We compile these software projects to 4 instruction set architectures including x86, x64, MIPS, and ARM, each with 4 optimizations, *i.e.,* `O0`-`O3`, using `GCC-7.5`, and 3 obfuscation strategies, including bogus control flow (`bcf`), control flow flattening (`cff`), and instruction substitution (`sub`), using Hikari [104] based on `Clang-8`. Table 1 summarizes the statistics of the datasets.

Pretraining and finetuning setup. We pretrain StateFormer (with GSM) on all datasets in Table 1. We sample a random 10% of the functions from the pretraining datasets as the validation set. We then pretrain the model in 10 epochs and checkpoint the model weights that achieve the lowest validation loss for finetuning. Note that GSM pretraining task *does not have any access to ground truth*

Table 2: StateFormer's precision, recall, and F1 score, for each architecture (ARCH), optimization (OPT), and obfuscation (OBF).

ARCH	OPT/OBF	Precision	Recall	F1 score
ARM	O0	77.1	79.2	78.1
	O1	78	76.2	77.1
	O2	77.3	73.7	75.4
	O3	90.9	89.9	90.4
MIPS	O0	98.9	91.7	95.2
	O1	86.1	67.6	75.7
	O2	80	68	73.4
	O3	81.3	71.2	75.8
x86	O0	85.3	83.8	84.5
	O1	72.4	70.9	71.6
	O2	74.8	70.9	72.8
	O3	83.6	79.8	81.6
x64	O0	81.5	81.4	81.4
	O1	75.8	74	74.9
	O2	71.1	69.2	70.1
	O3	72.3	70.4	71.3
	bcf	73.5	70.5	72
	cff	73.2	71.1	72.1
	sub	75.6	69.1	72.2

type labels. Therefore, we can always collect arbitrary binaries for pretraining, including those used in finetuning for type inference. This is a common practice in transfer learning [24, 67].

We finetune on StateFormer our dataset categorized by the architecture and optimization/obfuscation (Section 5.1). We partition the training and testing set by randomly selecting 90% of the functions for training and the remainder for testing.

Hyperparameters. We pretrain and finetune StateFormer for 10 epochs and 50 epochs, respectively. We set the default masking percentage $P_{mask} = 0.8$ (Section 3.3) and study different choices of P_{mask} in Section 5.4. We choose $\alpha = 4$ in Equation 1 such that the MSE of predicting μDataState and the BCE of predicting μControlState are scaled to the same magnitude. We perform an extensive evaluation of the properties of NAU to understand its capability in encoding numerical values and learning arithmetics. Due to the space constraints, we put the details of this study and the complete hyperparameter settings in our supplementary material.

5 EVALUATION

We aim to answer the following research questions.

- **RQ1**: How accurate is StateFormer in type inference?
- **RQ2**: How does StateFormer compare to the state-of-the-art ML-based systems?
- **RQ3**: How fast is StateFormer compared to other tools?
- **RQ4**: How effective is pretraining with GSM in improving the type inference accuracy?
- **RQ5**: How well does StateFormer approximate the operational semantics by training with GSM?

5.1 RQ1: Accuracy

We first study the accuracy of StateFormer on all binaries. Following the setup described in Section 4, we report the results in Table 2. StateFormer achieves an average 77.9% F1 score across all architecture, optimization, and obfuscation.

On x86 and x64, We observe that StateFormer remains relatively robust for binaries with higher optimization and obfuscation.

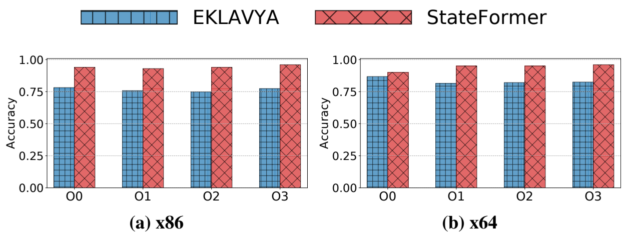


Figure 5: Accuracy of EKLAVYA and StateFormer on binaries of different architectures and optimizations.

For example, the F1 score for x86 `O3` is only 2.9% lower than that of x86 `O0`. The F1 score for x64 `O3` is only 3.6% lower than that of x64 `O1`. Regarding the performance across different architectures (with all optimizations/obfuscations), we notice no significant difference on average. These observations indicate that StateFormer is robust across architectures and optimizations with disparate operational semantics of their instructions.

StateFormer achieves an average 77.9% F1 score across all architecture, optimization, and obfuscation and remains robust for binaries with higher optimization levels and obfuscations.

5.2 RQ2: Comparison to Baseline

Baseline comparison. We compare StateFormer with 3 state-of-the-art type inference tools, namely EKLAVYA, Debin, and TypeMiner, as described in Section 4.

To compare with EKLAVYA, we evaluate StateFormer on the same 8 projects considered in their paper: Binutils, Coreutils, Findutils, sg3-utils, util-linux, Inetutils, Diffutils, and usbutils. We evaluate StateFormer on 7 types considered in EKLAVYA. EKLAVYA treats type inference for each argument (of multiple function arguments) as an independent classification task and reports the accuracy (instead of F1 score). We thus also evaluate StateFormer's accuracy, defined as the number of correctly predicted types divided by all the number of tokens.

Figure 5 compares StateFormer to EKLAVYA side-by-side on two architectures (*i.e.,* x86 and x64) and 4 optimizations (`O0`-`O3`) as EKLAVYA is evaluated with these settings. On average, StateFormer outperforms EKLAVYA by 13.3%. Notably, StateFormer remains robust across different optimization levels, while EKLAVYA has a clear drop when the optimization level is increased.

To compare with Debin, we run their released model on OpenSSL, which we have confirmed is not included in their training set. We compile OpenSSL into 3 architectures (x86, x64, and ARM) with 4 optimizations (`O0`-`O3`). As Debin considers only 17 types, we also restrict the prediction of StateFormer to the same 17 types. Figure 6 shows that StateFormer consistently outperforms Debin on all architectures and optimizations, achieving 14.6% higher F1 scores on average. We observe Debin has an apparent drop in F1 scores with higher optimizations (down to 46.1% for ARM), while StateFormer remains robust with at least 70% F1 scores.

Finally, we compare StateFormer to TypeMiner on the same datasets they have considered. We restrict our test on x64 with `O3`, as TypeMiner is evaluated only on x64. TypeMiner treats type inference

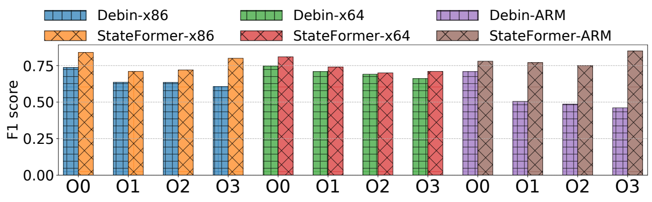
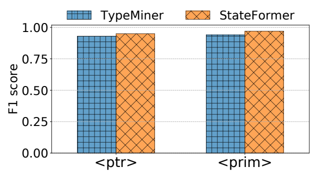
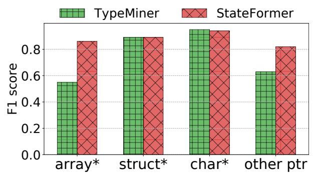


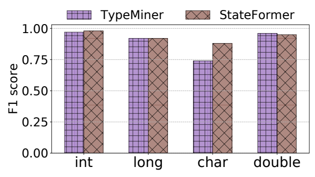
Figure 6: F1 of Debin and StateFormer in recovering types for binaries of different architectures and optimizations.



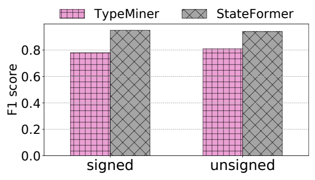
(a) Task 1



(b) Task 2



(c) Task 3



(d) Task 4

Figure 7: StateFormer's and TypeMiner's F1 scores in 4 type inference tasks defined in Section 5.2.

as a multi-stage classification task, training independent classifiers to predict types at different levels. For example, it first trains a binary classifier to predict whether a variable is a pointer or not and then trains a second classifier to predict the pointer type. Since they do not make complete predictions in one-shot, we compare StateFormer on 4 sub-tasks on which TypeMiner has been evaluated. Specifically, TypeMiner's first prediction task is a binary classification task deciding whether a variable has a pointer type (`<ptr>`) or a primitive type (`<prim>`). Its second task is to predict the pointer types, including `array*`, `struct*`, `char*`, and `other ptr`. Its third task is to predict the primitive types, including `int`, `long int`, `char`, and `double`. Its fourth task is to predict the signedness, including `signed` and `unsigned`. We label these 4 tasks as Task 1-4.

Figure 7 demonstrates that StateFormer outperforms TypeMiner in 4 tasks by an average 8.2%. In particular, TypeMiner significantly fluctuates when predicting primitive types (Task 3) and pointer types (Task 2), but StateFormer is more robust.

StateFormer outperforms EKLAVYA, Debin, and TypeMiner by 13.3%, 14.6%, and 8.2%, respectively, and is more robust than all baselines for different optimizations and type granularity.

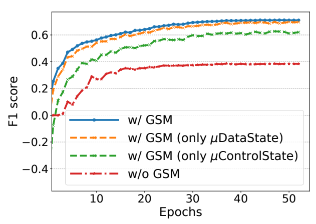
5.3 RQ3: Inference Speed

We evaluate StateFormer's inference speed on binary programs and compare it to Debin and Ghidra. Specifically, we consider 4 software projects with different sizes on x64 compiled with `O0`.

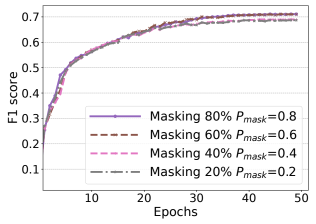
Table 3: Execution time (in seconds) of StateFormer (on both CPU and GPU), Debin, and Ghidra on 4 of our datasets, with diverse number of instructions (measured in thousand).

Project	# Inst (k)	Runtime				StateFormer Speedup
		StateFormer		Debin	Ghidra	
		CPU	GPU			
ImageMagic	1,252	187.8	7.3	N/A*	664.3	91×
PuTTY	969	146.0	5.6	5239.8	514.2	91.8×
Findutils	157	23.7	0.9	849.0	83.3	92.6×
zlib	22	3.3	0.1	119.0	11.7	117×

*Debin terminates abruptly after running one of the binaries for 138 minutes.



(a) Pretraining GSM effect



(b) Masking percentage effect

Figure 8: (Left) StateFormer's testing F1 scores when it is (1) pretrained with GSM, (2) pretrained with only predicting μDataState, (3) pretrained with only predicting μControlState, or (4) not pretrained. (Right) StateFormer's validation F1 score at each finetuning epoch when the masking percentages P_{mask} in GSM are 0.8, 0.6, 0.4, or 0.2.

Table 3 shows the runtime performance of StateFormer, Debin, and Ghidra. StateFormer (based on GPUs) achieves 98.1× speedup on average than the second-best tool. Notably, while the authors of Debin have tried to optimize their underlying learning algorithms (conditional random field) with parallelized implementation [78], it performs 1023× and 35.8× slower than StateFormer GPU and CPU, respectively. We attribute the speedup of StateFormer to its underlying neural architecture, which is amenable to GPU acceleration, while neither Debin's nor Ghidra's underlying algorithms can be implemented using GPU efficiently.

StateFormer is 98.1× faster than the second-best tool.

5.4 RQ4: Effectiveness of GSM

In this section, we dig deeper into the effectiveness of GSM pretraining task by quantifying how much improvement that StateFormer achieves when pretrained with GSM.

Effectiveness of GSM. We compare StateFormer's finetuning accuracy when it is (1) pretrained with GSM, (2) pretrained with *partial* GSM by only predicting μDataState, (3) pretrained with *partial* GSM by only predicting μControlState, and (4) not pretrained.

Figure 8a shows StateFormer's validation F1 score at each finetuning epoch. It clearly demonstrates that StateFormer pretrained with complete GSM achieves the best finetuning F1 scores: it reaches 71.3% F1 score within the 50 epochs. Without pretraining with GSM, it only achieves 38.3% F1 score. We also note that StateFormer pretrained with only predicting μDataState outperforms that with only predicting μControlState. This is intuitive as predicting μDataState requires understanding instructions' actual execution effect and computing the concrete values, while predicting μControlState is only a binary classification task encoding approximate control flow. Nevertheless, we observe even pretraining with predicting only μDataState or μControlState is still beneficial for type inference, as STATEFORMER pretrained on either of them obtains 69% and 62% F1 scores, respectively.

Masking percentage. Recall in GSM, we train STATEFORMER to reconstruct the masked μDataState, and we use default masking percentage $P_{mask} = 0.8$ throughout our experiments (Section 4). As masking less percentage of μDataState makes it easier to train on GSM, we study how varying P_{mask} affects the type inference performance. Figure 8b shows the validation F1 scores achieved by STATEFORMER when we vary P_{mask}. We observe that the more we mask in GSM, the better it boosts the type inference performance, but the gap of improvement is not significant. One possible explanation is that even in one example, the masked states are less, many pretraining samples and the dynamic masking still introduce diverse enough cases for learning operational semantics.

STATEFORMER pretrained with GSM outperforms that without pretraining by 33% in F1 score. Masking percentage in pretraining GSM does not significantly affect the finetuning results: pretraining with 20% masking rate results in <2% decrease in F1 score compared to pretraining with 80% masking rate.

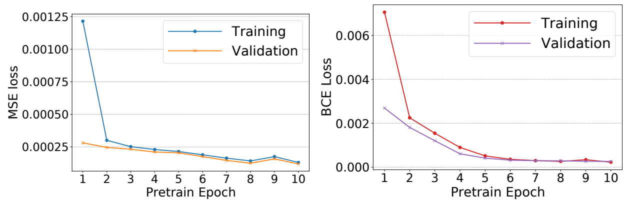
5.5 RQ5: STATEFORMER Performance on GSM

Pretraining losses with GSM. We also study the losses of pretraining STATEFORMER with GSM. Such a study directly validates whether pretraining with GSM indeed helps STATEFORMER to learn operational semantics. Low losses on unseen testing μState and function binaries indicates that STATEFORMER highly likely learns to generalize based on its learned knowledge of operational semantics.

Figure 9 shows the training and validation losses in 10 epochs of pretraining STATEFORMER with GSM. The validation set is constructed by sampling a random 10% functions from the projects used in pretraining (as described in Section 4). Specifically, Figure 9a shows the MSE loss of predicting μDataState and Figure 9b shows the BCE loss of predicting μControlState. We observe that the validation MSE drops to 0.00011, which translates to average absolute distance (by taking the square root) between prediction and groundtruth as 0.011 ($\sqrt{0.00011} = 0.011$). As we normalize the byte values from $[0, 256]$ into $[0, 1]$ (see Section 3.2), $0.011 \times 256 = 2.8$ is the actual absolute error between the predicted byte and the groundtruth. The average error within the deviation of only 3-byte indicates that STATEFORMER learns to approximate the execution effect.

Effects of control and data flow pretraining. Concurrent to our work, Trex [68] also leverages transfer learning to learn program execution semantics. However, Trex completely ignores control and data flow modeling as it focuses on binary similarity detection. In contrast, STATEFORMER focuses on type inference; therefore, it requires precise data flow (type of output of an instruction depends on types of operands) and control flow (it must also infer types of values in the unexecuted portion of the code).

Because of these differences in the high-level requirements of the downstream tasks, STATEFORMER and Trex adopt significantly



(a) MSE in predicting μDataState **(b) BCE in predicting μControlState**

Figure 9: MSE and BCE of predicting μDataState and μControlState, respectively, during pretraining STATEFORMER with GSM.

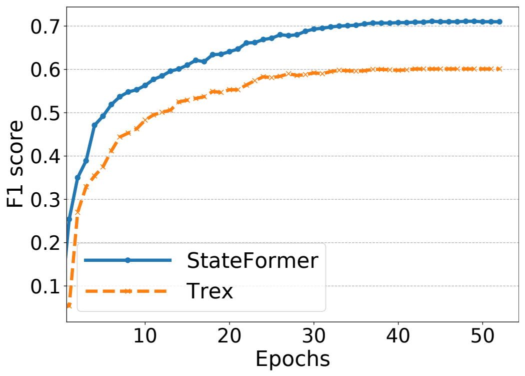


Figure 10: Type inference F1 score between models pretrained by GSM and Trex's pretraining objective.

different pretraining approaches, *i.e.,* generating control and data flow state (GSM) vs. code and trace token classification. In general, it remains an open challenge in transfer learning to determine which pretraining task is the most effective for which downstream task. Part of our contribution in STATEFORMER is to design a pretraining task that makes the downstream task of type inference precise. For example, Figure 10 shows that STATEFORMER outperforms Trex by around 10.9 percentage points in F1 score for type inference.

Probing STATEFORMER on real-world code. Besides quantifying the pretraining losses, we probe the pretrained STATEFORMER using a concrete binary example to study how it predicts μState.

Consider the example in Figure 11. We examine how STATEFORMER predicts registers `esp`, `edi`, `ebp`, and `esi` from input μDataState, in which we mask all registers except for their first appearances. The accurate prediction of `esp` at line 5 suggests that STATEFORMER is able to associate `0x0886644e` with `esp` at line 1 and line 2 and understand the execution effect of `sub`. Further, to predict `esi` at line 6, STATEFORMER needs to understand `xor`'s execution effects at line 3. Since there is no other occurrence of `esi` in this code block, we can conclude that the prediction of `esi` is based solely on STATEFORMER's understanding of `xor`.

GSM is effective in assisting STATEFORMER to learn various instructions' operational semantics. STATEFORMER's absolute error in predicting μDataState during pretraining GSM is very low (within 3 on average for each byte).

```
<remove_quoted_ifs>:

Code:                       Input  μDataState:
...                         ...
1  mov ebp,esp              mov 0x0885544e,0x0886644e
2  sub esp,hexv             sub <mask>,0x00000000
...                         ...
3  xor esi,esi              xor 0x0886644f,0x0886644f
4  mov edi,eax              mov 0x02222883,0x00000000
5  mov [esp],edi            mov [<mask>],<mask>
...                         ...
6  mov [ebp-hexv],esi       mov [<mask>],<mask>
```

Line Number	Register	Ground Truth	Prediction
2	**esp**	0x0886644e	0x0886644e
5	**esp**	0x0886644e	0x0886644e
5	**edi**	0x00000000	0x00000000
6	**ebp**	0x0886644e	0x0886644e
6	**esi**	0x00000000	0x00000000

Figure 11: The code and μDataState trace from `remove_quoted_ifs` in `bash`. We highlight with same colors the masked values and locations that StateFormer relies on to make the prediction.

6 THREATS TO VALIDITY

Target binaries. We focus on the binary ready to be disassembled and do not consider maliciously encrypted code or packed binaries as it requires an entirely different toolchain to unpack. StateFormer can be applied once the binaries are unpacked or decrypted.

Datasets. We aim to collect diverse datasets of software projects to expose various instances of operational semantics. To this end, we ensure our datasets have different implemented functionalities (*e.g.,* utility functions, image processing functions, etc.).

Hyperparameters. We keep most hyperparameters fixed throughout the evaluation, considering the fact that there is no principled method for tuning hyperparameters to date [50]. Nevertheless, we ensure our hyperparameter choice is empirically reasonable (Section 4).

7 RELATED WORK

There are two main lines of prior works that are related to our work – type inference from binaries (*e.g.,* for binary hardening and decompilation) and type inference from source code of dynamically-typed languages (*e.g.,* for software debugging, IDE support, and API understanding for developers) [42, 73, 95]. In this paper, we focus on type inference for binaries. Binary analysis is known to be more challenging as recovering stripped source-level constructs is an undecidable problem [62, 66]. Moreover, many source-level type hints such as the variable name and the computation that operates on the variable are absent at binary-level. We summarize common approaches used for different type inference tasks below.

Traditional approaches. Static analysis has been widely adopted in off-the-shelf reverse engineering tools for type inference [1, 14, 81, 82, 85, 86]. A standard static analysis approach for type inference uses domain-expert-provided rules for different instructions/statements to either directly specify the operand types [6, 23, 31, 32, 39, 44], or define how types should be propagated from instructions with known types to other instructions [9, 19, 21, 25, 28, 51, 52, 55, 103]. To track the type propagation, these works often rely on expensive data/control dependency analysis [8, 9, 46, 47, 51, 52, 59, 72, 82].

By contrast, dynamic analysis uses accurate program states and memory access patterns observed during program execution [27] to define precise rules for type inference [5, 15, 22, 38, 48, 77, 79, 84] and propagation [17, 34–36, 54, 83, 102]. However, dynamic approaches suffer from low code coverage, leading to a high false negative rate [51]. Increasing code coverage requires collecting and combining dynamic traces from multiple program executions [15, 83], which incurs prohibitively high overhead.

StateFormer enjoys the benefits of both static and dynamic analysis as it *automates* learning instructions' approximate operational semantics from *cheap micro-execution* and uses such semantics to learn type inference rules *without dynamic execution*.

ML-based approaches. Recently, machine learning has been increasingly applied to type inference. Examples include inferring the type of function argument [20], recovering general variable type [41, 59, 60, 71, 97, 100], and other metadata (*e.g.,* variable names) [2, 40, 56, 78, 80, 90, 92, 94]. However, existing ML-based binary type inference approaches use only static code without any traces and suffer from similar limitations as static analysis. Concurrent to our work, Trex [68] also leverages transfer learning to learn program execution semantics. However, Trex is not control/data flow aware, resulting in a significant performance drop in the type inference task, as shown in Section 5.5.

More broadly, machine learning has shown significant success in learning generalizable representation that applies to many program analysis tasks [3, 13, 69]. StateFormer contributes a new generic framework to learn programs' operational semantics. Therefore, we believe StateFormer has a great potential to apply to other downstream program analysis tasks beyond type inference.

8 CONCLUSION

We presented StateFormer, a neural architecture that uses the operational semantics of assembly code to recover type information from stripped binaries. We designed a novel pretraining task, Generative State Modeling, to help StateFormer to learn code operational semantics and transfers this knowledge to learn type inference rules. We showed that StateFormer is 14.6% more accurate than state-of-the-art tools, and our ablation studies showed that GSM improves type inference accuracy by 33%.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF grants CNS-1842456, CNS-1801426, CNS-1617670, CCF-1822965, CCF-1845893, CNS-1563843, CNS-1564055, and IIS-2040961; ONR grants N00014-17-1-2010, N00014-16-1-2263, and N00014-17-1-2788; an NSF CAREER award; an ARL Young Investigator (YIP) award; a Google Faculty Fellowship; a J.P. Morgan Faculty research award; a DiDi Faculty research award; a Google Cloud grant; a Capital One research grant; and an Amazon Web Services grant. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, ARL, NSF, Captital One, Google, J.P. Morgan, DiDi, or Amazon.

REFERENCES

[1] National Security Agency. 2019. Ghidra Disassembler. https://ghidra-sre.org/.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *2015 10th Joint Meeting on Foundations of Software Engineering*.

[3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *Comput. Surveys* (2018).

[4] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. 2020. BinRec: dynamic binary lifting and recompilation. In *Fifteenth European Conference on Computer Systems*.

[5] Jong-hoon An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby. *ACM SIGPLAN Notices* (2011).

[6] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards type inference for JavaScript. In *European conference on Object-oriented programming*.

[7] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy*.

[8] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*.

[9] Gogul Balakrishnan and Thomas Reps. 2007. Divine: Discovering variables in executables. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*.

[10] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Security Symposium*.

[11] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*.

[12] Eli Bendersky. [n.d.]. PYEFLTOOLS. https://github.com/eliben/pyelftools.

[13] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Programming with" big code": Lessons, techniques and applications. In *1st Summit on Advances in Programming Languages*.

[14] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*.

[15] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. 2010. Binary Code Extraction and Interface Identification for Security Applications. In *2010 Network and Distributed System Security Symposium*.

[16] Juan Caballero and Zhiqiang Lin. 2016. Type inference on executables. *Comput. Surveys* (2016).

[17] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *16th ACM conference on Computer and communications security*.

[18] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *2015 Network and Distributed System Security Symposium*.

[19] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. 2005. String analysis for x86 binaries. In *6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*.

[20] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium*.

[21] Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[22] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. 2008. Digging for Data Structures. In *2008 USENIX Symposium on Operating Systems Design and Implementation*.

[23] Loris d'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin Pierce. 2013. Sensitivity analysis using type-based constraints. In *1st annual workshop on Functional programming concepts in domain-specific languages*.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[25] David Dewey and Jonathon T Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code. In *2012 Network and Distributed System Security Symposium*.

[26] EN Dolgova and AV Chernov. 2009. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software* (2009).

[27] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *34th ACM SIGPLAN conference on Programming language design and implementation*.

[28] MV Emmerik and Trent Waddington. 2004. Using a decompiler for real-world source recovery. In *11th Working Conference on Reverse Engineering*.

[29] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *2003 International Conference on Software Engineering Workshop on Dynamic Analysis*. 24–27.

[30] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the effectiveness of type-based control flow integrity. In *34th Annual Computer Security Applications Conference*.

[31] Alexander Fokin, Katerina Troshina, and Alexander Chernov. 2010. Reconstruction of class hierarchies for decompilation of C++ programs. In *2010 14th European Conference on Software Maintenance and Reengineering*.

[32] Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. 2009. Static type inference for Ruby. In *2009 ACM symposium on Applied Computing*.

[33] Patrice Godefroid. 2014. Micro execution. In *36th International Conference on Software Engineering*.

[34] Neville Grech, Bernd Fischer, and Julian Rathke. 2018. Preemptive type checking. *Journal of logical and algebraic methods in programming* (2018).

[35] Neville Grech, Julian Rathke, and Bernd Fischer. 2013. Preemptive type checking in dynamically typed languages. In *International Colloquium on Theoretical Aspects of Computing*.

[36] Philip J Guo, Jeff H Perkins, Stephen McCamant, and Michael D Ernst. 2006. Dynamic inference of abstract types. In *2006 international symposium on Software testing and analysis*.

[37] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2016. TypeSan: Practical type confusion detection. In *2016 ACM SIGSAC Conference on Computer and Communications Security*.

[38] Istvan Haller, Asia Slowinska, and Herbert Bos. 2013. Mempick: High-level data structure detection in C/C++ binaries. In *2013 20th Working Conference on Reverse Engineering*.

[39] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *International Conference on Computer Aided Verification*.

[40] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. DEBIN: Predicting debug information in stripped binaries. In *2018 ACM SIGSAC Conference on Computer and Communications Security*.

[41] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*.

[42] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium*.

[43] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. 2018. Dependence-preserving data compaction for scalable forensic analysis. In *27th USENIX Security Symposium*.

[44] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*.

[45] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *27th ACM SIGSOFT international symposium on software testing and analysis*.

[46] Wuxia Jin, Yuanfang Cai, Rick Kazman, Gang Zhang, Qinghua Zheng, and Ting Liu. 2020. Exploring the Architectural Impact of Possible Dependencies in Python Software. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering*.

[47] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. 2014. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*.

[48] Changhee Jung and Nathan Clark. 2009. DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[49] Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining. In *33rd Conference on Neural Information Processing Systems*.

[50] Yann LeCun, Patrice Y Simard, and Barak Pearlmutter. 1993. Automatic learning rate maximization by on-line estimation of the hessian's eigenvectors. In *1993 Advances in Neural Information Processing System*.

[51] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. In *2011 Network and Distributed System Security Symposium*.

[52] Junghee Lim, Thomas Reps, and Ben Liblit. 2006. Extracting output formats from executables. In *2006 13th Working Conference on Reverse Engineering*.

[53] Yan Lin and Debin Gao. 2021. When Function Signature Recovery Meets Compiler Optimization. In *2021 IEEE Symposium on Security and Privacy*.

[54] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *2008 Network and Distributed System Security Symposium*.

[55] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *2010 Network and Distributed System Security Symposium*.

[56] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering*.

[57] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *2019 ACM SIGSAC Conference on Computer and Communications Security*.

[58] Andreas Madsen and Alexander Rosenberg Johansen. 2020. Neural Arithmetic Units. In *International Conference on Learning Representations*.

[59] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

[60] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering*.

[61] James Martens and Ilya Sutskever. 2011. Learning recurrent neural networks with hessian-free optimization. In *28th international conference on machine learning*.

[62] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *41st International Conference on Software Engineering*.

[63] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI Conference on Artificial Intelligence*.

[64] Hanne Riis Nielson and Flemming Nielson. 1992. *Semantics with applications*. Vol. 104. Springer.

[65] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. Fairseq: A fast, extensible toolkit for sequence modeling. In *2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Demonstrations*.

[66] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *2021 IEEE Symposium on Security and Privacy*.

[67] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *2021 Network and Distributed System Security Symposium*.

[68] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv preprint arXiv:2012.08680* (2020).

[69] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*.

[70] G. D. Plotkin. 1981. A Structural Approach to Operational Semantics. *University of Aarhus* (1981).

[71] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[72] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

[73] Michael Pradel and Koushik Sen. 2015. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th European Conference on Object-Oriented Programming*.

[74] Aravind Prakash, Heng Yin, and Zhenkai Liang. 2013. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *8th ACM SIGSAC symposium on Information, computer and communications security*.

[75] Nguyen Anh Quynh. 2014. Capstone: Next-gen disassembly framework. *Black Hat USA* (2014).

[76] NGUYEN Anh Quynh and DANG Hoang Vu. 2015. Unicorn: Next Generation CPU Emulator Framework. *BlackHat USA* (2015).

[77] Easwaran Raman and David I August. 2005. Recursive data structure profiling. In *2005 workshop on Memory system performance*.

[78] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "big code". *ACM SIGPLAN Notices* (2015).

[79] Brianna M Ren, John Toman, T Stephen Strickland, and Jeffrey S Foster. 2013. The ruby type checker. In *28th Annual ACM Symposium on Applied Computing*.

[80] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2012. Automated API property inference techniques. *IEEE Transactions on Software Engineering* (2012).

[81] Hex-Rays SA. 2008. IDA Pro Disassembler.

[82] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy*.

[83] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *2011 Network and Distributed System Security Symposium*.

[84] Venkatesh Srinivasan and Thomas Reps. 2014. Recovery of class hierarchies and composition relationships from machine code. In *International Conference on Compiler Construction*.

[85] Binary Ninja Team. 2015. Binary Ninja – A new type of reversing platform. https://binary.ninja/.

[86] Radare2 Team. 2017. Radare2 GitHub repository. https://github.com/radare/radare2.

[87] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-sensitive pointer analysis for symbolic execution. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[88] Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. 2018. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*. 8035–8044.

[89] David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. 2014. Sigpath: A memory graph based approach for program data introspection and modification. In *European Symposium on Research in Computer Security*.

[90] Muhammad Usman, Wenxi Wang, Kaiyuan Wang, Cagdas Yelen, Nima Dini, and Sarfraz Khurshid. 2020. A study of learning likely data structure properties using machine learning models. *International Journal on Software Tools for Technology Transfer* (2020).

[91] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy*.

[92] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *2017 11th joint meeting on foundations of software engineering*.

[93] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *2017 Advances in Neural Information Processing Systems*.

[94] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*.

[95] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2009. Locating need-to-translate constant strings for software internationalization. In *2009 IEEE 31st International Conference on Software Engineering*.

[96] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *28th Annual Computer Security Applications Conference*.

[97] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. In *2020 International Conference on Learning Representations*.

[98] Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *IEEE* 78, 10 (1990), 1550–1560.

[99] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[100] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *24th ACM SIGSOFT international symposium on foundations of software engineering*.

[101] Dongrui Zeng and Gang Tan. 2018. From Debugging-Information Based Binary-Level Type Inference to CFG Generation. In *Eighth ACM Conference on Data and Application Security and Privacy*.

[102] Junyuan Zeng, Yangchun Fu, Kenneth A Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation resilient binary code reuse through trace-oriented programming. In *2013 ACM SIGSAC conference on Computer & communications security*.

[103] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity. In *2015 Network and Distributed System Security Symposium*.

[104] Naville Zhang. 2017. Hikari – an improvement over Obfuscator-LLVM. https://github.com/HikariObfuscator/Hikari.